Proceedings of the 51st Hawaii International Conference on System Sciences | 2018

ASAS: An Approach to Support Simulation of Smart Systems

Valdemar Vicente Graciano Neto*†‡, Lina Garcés*†, Milena Guessi*†, Carlos Eduardo B. Paes§,
Wallace Manzano*, Flavio Oquendo† and Elisa Nakagawa*
*Universidade de São Paulo (USP), São Carlos - SP - Brazil
†Université de Bretagne Sud (UBS), Vannes, Bretagne – France
‡Universidade Federal de Goiás (UFG), Goiânia - GO – Brazil
§Pontifícia Universidade Católica de São Paulo (PUC-SP), São Paulo – SP – Brazil
Emails: valdemarneto@usp.br, linamgr@icmc.usp.br, milena@icmc.usp.br, carlosp@pucsp.br,
wallace.manzano@usp.br, flavio.oquendo@irisa.fr, elisa@icmc.usp.br

***Abstract*—Smart systems, such as smart cities, smart buildings, and autonomous cars, have recently gained increasing popularity. Each such system is essentially a System-of-Systems (SoS). SoS are dynamically established as alliances among independent and heterogeneous software systems to offer complex functionalities as a result of constituents interoperability. An SoS often supports critical application domains, and, as such, must be reliable. Many SoS have been specified and evaluated for their correct operation using static models. However, specification languages have not supported to capture their inherent dynamic nature nor enabled to monitor their operation. The main contribution of this paper is to present ASAS, an approach to Automatically generate Simulation models for smArt Systems (ASAS) in order to support evaluation of their operation. In particular, our approach makes it possible to transform formal models of the SoS architecture (expressed in SoSADL) into simulation models (expressed in DEVS). We evaluated our approach by conducting two case studies using a flood monitoring system that is intended to be part of a smart city. Results indicate that ASAS can successfully generate functional simulations for the SoS operation, which in turn can enable to reason and monitor an SoS operation, taking into account its dynamic nature.**

***Keywords*-System-of-Systems; Simulation; Model-Based Engineering.**

I. Introduction

Smart systems such as smart cities are highly dynamic systems in which software influences their entire life cycle [1, 10]. New systems can dynamically join a smart city such as cars that come from other cities and join the traffic, or people that go to work and connect their mobile phones to the smart city network. Smart systems offer unique functionalities as a result of their interoperability, such as optimizing long traffic queues when it is raining or calculating alternative routes to avoid traffic jam after accidents. Such smart systems are broadly known as Systems-of-Systems (SoS)[1], i.e., a set of operationally and managerially independent software-intensive systems (known as constituents) that work together to achieve complex functionalities [11]. Besides individual systems behaviors, SoS exhibit emergent behaviors, which are dynamic capabilities that emerge at runtime as a result of the interoperability among constituents [11], and dynamic architecture, i.e., an architecture in which constituents and the SoS structure itself can change at runtime [7]. Due to critical domains where SoS can be found, faults can cause disasters, financial loss, or even deaths. Hence, SoS must be trustworthy, that is, they must work in a reliable way without failures [7]. To achieve such trustworthiness, SoS must be evaluated in regards to their dynamic operation, adopting models that can precisely capture and represent SoS [13].

Modeling languages (such as UML[2], SysML[3], and CML[4]) support both representation and validation of static properties of the SoS. However, they lack mechanisms to capture SoS dynamics, hampering evaluation of the operation of SoS [8]. Simulations provide the means to evaluate SoS operation by anticipating failures and unveiling dynamic aspects of systems' operation [5, 13, 17]. Therefore, we pose the following research question: *Is it possible to automatically generate simulation models for a SoS to support evaluating its operation*? The main contribution of this paper is to present ASAS, an approach that automatically transforms static SoS models (expressed in a language termed SoSADL) into simulation models (expressed in DEVS - Discrete Event System Specification). We applied our approach in two independent case studies that automatically generated simulation models for a Flood Monitoring and Emergency Management SoS (FMSoS). Preliminary results show confidence in our approach, which was able to successfully transform abstract descriptions of this SoS into functional simulation models, leveraging constituents individual capabilities and reinforcing SoS trustworthiness. The remaining of this paper is structured as follows: Section II presents foundations; Section III provides details of ASAS; Section IV presents the two case studies; and Section V concludes the paper, along with directions for future work.

[1]For sake of simplicity, SoS is used interchangeably to express singular and plural.

[2]UML, http://www.uml.org/
[3]SysML, http://sysml.org/
[4]CML, http://www.compass-research.eu/approach.html

URI: http://hdl.handle.net/10125/50613
ISBN: 978-0-9981331-1-9

II. Background

SoS are being developed for critical domains in the management of smart cities. Specific means of predicting SoS operation become necessary to offer reliable services for SoS stakeholders [15]. SoS exhibit dynamic architectures, i.e., architectures that can assume diverse architectural configurations (i.e., also known as coalitions) during SoS runtime, due to addition, substitution, and deletion of constituents, or reorganization of the structure of the SoS architecture [3].

SoSADL is a novel formal language that has been created to support the specification of SoS architectures [6, 15]. It supports the specification of abstract architectures in which constituents are known at design-time and abstract connectors (also referred to as mediators) that can be dynamically realized for composing concrete architectures of this SoS. Mediators are first-class elements representing communication links between two or more constituents [18]. The architecture of a SoS defines policies for assembling abstract types of systems and mediators as coalitions, which are further characterized by behavior, data type, and gate declarations. Gates are abstractions that enable the establishment of connections. A connection can receive stimulus from or act on the environment, enabling the communication among independent elements. Data types can have inherent functions, and functions can be associated to expressions. Mediators and systems can be also specified in terms of gates, data types, and behaviors. Despite their expressiveness, SoSADL models are not yet executable. Hence, a dynamic view of architectural descriptions expressed in this language is missing.

In parallel, simulations can be central for engineering SoS. DEVS is a formalism that can support SoS simulation [20]. DEVS is based on atomic and coupled models. Atomic models can represent individual entities, such as constituent systems and mediators, and coupled models can represent a combination of atomic models, such as coalitions. Atomic models comprise the following elements: (i) ports (input and output); (ii) a labeled state diagram that executes transitions in response to input and output events (hence governing the system operation); (iii) functions that can be used to process data; (iv) data types; and (v) events. Coupled models are expressed as a System Entity Structure (SES), a formal structure governed by a small number of axioms that expresses how atomic models communicate with each other [20]. Moreover, DEVS supports dynamic reconfiguration.

Related work. Several initiatives have been proposed to address SoS representation and evaluation considering: 1) support for Software Architectural Description of SoS, including constituents not totally known at design-time; 2) support for representation and/or evaluation of SoS dynamic architectures; and 3) support for SoS Simulation.

Table I shows a comparison among approaches related to ASAS. Approaches highlighted in bold font are model-based approaches, i.e., they use abstract models representing the software system and make one or more model transformations for generating a target model. The bi-graph approach offers a mathematical solution for representing and simulating emergent behaviors in SoS [17]. However, this approach lacks the notion of software architectures. Cavalcante et al. [2] report on a model transformation approach that adopts π-ADL architectural models to describe dynamic architectures, transforming π-ADL models into Go software code. However, π-ADL does not provide straightforward abstractions for some particular concepts of SoS, such as mediators, coalitions and constituents that are not known at design-time. DEVS is an approach to deal with the dynamic properties of SoS [20]. In its standard form, it relies on systems engineering concepts, which combine hardware concepts and low-level abstractions, such as input/output and state diagram procedures. Falkner et al. [4] propose a change of emphasis from SoS specifications to executable models for the purposes of performance prediction. However, their proposal mentions drawbacks related to: (i) use of an informal description of the SoS architecture with a notation based on UML, and (ii) lack of dynamic and functional properties. Xia et al. [19] evaluated the SoS architectures documented in DoDAF[5] by means of a model-driven approach that transforms system architecture models in Simulink[6] into executable models. Their approach focuses on measuring non-functional properties, such as feasibility and efficiency, but it does not consider software architectures.

Table I: Comparison among related approaches

Approach	(1)	(2)	(3)
Bi-graph Approach [17]	✗	✗	✓
Cavalcante et al. [2]	✗	✗	✓
DEVS [20]	✗	✓	✓
Falkner et al. [4]	✗	✗	✓
Xia et al. [19]	✗	✓	✓
ASAS	✓	✓	✓

III. From Software Architectures to Simulation Models

We present a model-based approach termed ASAS, which automatically generates simulation models to evaluate SoS operation. This approach implements an automated transformation from SoS architecture models expressed in SoSADL into dynamic models expressed in DEVS. Moreover, this transformation is accomplished in two parts: one generates atomic models and the other generates coupled models.

Our approach encompasses five steps, which are: (i) specification of an SoS software architecture using SoSADL; (ii) transformation of the architecture description into a functional DEVS simulation model (automatically accomplished

[5]The DoDAF Architecture Framework Version 2.02. US. Department of Defense, Aug. 2010.

[6]http://www.mathworks.com/products/simulink/

by an Xtend[7] script); (iii) deployment of the simulation code in the MS4ME[8] environment; (iv) simulation of the SoS software architecture; and (v) analysis of the results.

A. Correspondences between SoSADL and DEVS models

Since DEVS and SoSADL are both founded upon rigorous formalizations, a transformation should preserve the correspondences between the fundamental concepts. We established the correspondences between both models as shown in Table II. We describe the mapping as follows:

Table II: Mapping between SoSADL and DEVS

SoS concept	SoSADL	DEVS
Connection	Connection Declaration	DEVS Port
Constituent System	System Declaration	Atomic Model
Data Types	Data Type Declaration	Data Type
Gate	Gate Declaration	DEVS Port
Mediator	Mediator Declaration	Atomic Model
Architecture	Coalition	Coupled Model

Connections. A connection can be established to receive stimuli or to act on the environment, enabling the communication among architectural elements. Translating to DEVS, connections are mapped into ports (input and output ports).
Constituent Systems. A system plays the role of a constituent that participates in the SoS. Constituents, become atomic models in DEVS.
Data Types. A data type can be specified for the scope of an SoS, constituent, or mediator. Data types and values in SoSADL become data types in DEVS.
Gates. A gate is an abstraction that enables to establish connections, and connections are abstractions of interactions exchanged between gates of independent entities. As gates do not have a direct mapping in DEVS, connections linked to gates are mapped as DEVS Ports, and gates are suppressed.
Mediators. Similarly to constituent systems, i.e., each mediator also becomes an atomic model.
Architectures. An architecture is transformed into a coupled model in DEVS that specifies how constituents interact in the coalition.

To illustrate how models are generated, we show a simplified version of a Flood Monitoring SoS (FMSoS), which is composed of abstract types of sensors, gateway, and transmitters.

B. Generating Atomic Models

Listing 1 shows a simplified code of a mediator type named Transmitter described in SoSADL. This element can be automatically transformed into an atomic model, which is depicted in Listing 2. Data types are defined on Lines 2-6. These data types are respectively transformed to datatypes in DEVS atomic model (i.e., lines 1-22 in Listing 2). Duties (which designate gates of mediators) with their respective connections are defined on Lines 8-16. The individual behavior of this mediator is specified in lines 18-23. In short, this behavior defines that after receiving the coordinates of constituents that it mediates (lines 19-20), this mediator will repeatedly receive data from one sensor (line 22) and forward them to the other (line 23). Similarly to gates, duties are transformed into inputs of ports in the atomic model (lines 24-27 in Listing 2). Finally, the behavior is translated into an automata in the atomic model (lines 29-42 in Listing 2).

```
1  mediator Transmitter( distancebetweengates:
       Distance ) is {
2    datatype Abscissa
3    datatype Ordinate
4    datatype Coordinate is tuple { x:Abscissa, y:
         Ordinate }
5    datatype Depth
6    datatype Measure is tuple { coordinate:
         Coordinate, depth:Depth }
7
8    duty transmit is {
9      connection fromSensors is in { Measure }
10     connection towardsGateway is out { Measure }
11   }
12
13   duty location is {
14     connection fromCoordinate is in { Coordinate }
15     connection toCoordinate is in { Coordinate }
16   }
17
18   behavior transmitting is {
19     via location::fromCoordinate receive
           coordinate
20     via location::toCoordinate receive coordinate
21     repeat {
22       via transmit::fromSensors receive measure
23       via transmit::towardsGateway send measure
24     }
25   }
26 }
```

Listing 1: Code in SoSADL for a mediator.

```
1  A Distance has a value!
2  the range of Distance's value is Integer!
3  use distance with type Distance!
4
5  A Abscissa has a value!
6  the range of Abscissa's value is Integer!
7  use abscissa with type Abscissa!
8  A Ordinate has a value!
9  the range of Ordinate's value is Integer!
10 use ordinate with type Ordinate!
11 Coordinate has x and y!
12 the range of Coordinate's x is Abscissa!
13 the range of Coordinate's y is Ordinate!
14 use coordinate with type Coordinate!
15
16 A Depth has a value!
17 the range of Depth's value is Integer!
18 use depth with type Depth!
19 Measure has coordinate and depth!
20 the range of Measure's coordinate is Coordinate!
21 the range of Measure's depth is Depth!
22 use measure with type Measure!
23
24 accepts input on FromCoordinate with type
       Coordinate!
25 accepts input on ToCoordinate with type Coordinate
       !
26 accepts input on FromSensors with type Measure!
27 generates output on Measure with type Measure!
28
29 to start hold in s0 for time 1!
30 hold in s0 for time 1!
31 from s0 go to s1! //Unobservable
32 passivate in s1!
```

[7]Xtend, xtend-lang.org/
[8]MS4ME, http://www.ms4systems.com/pages/ms4me.php

```
33  when in s1 and receive Coordinate go to s2!
34  passivate in s2!
35  when in s2 and receive Coordinate go to s3!
36  passivate in s3!
37  when in s3 and receive Measure go to s4!
38  hold in s4 for time 1!
39  after s4 output Measure!
40  from s4 go to s5!
41  hold in s5 for time 1!
42  from s5 go to s3! //Unobservable
```

Listing 2: An atomic model for a Mediator generated in DEVS.

Listing 3 shows part of the transformation code in Xtend that generates state transitions for the atomic model behavior based on elements specified in SoSADL. If the transformation consists of an input transition, the code in lines 2-7 is executed. If it consists of an output transition, the code in lines 9-15 is executed.

```
1  def compile(Element e) {
2  if ((connection.type == INPUT) or (action.type ==
       RECEIVE)) {
3    if(e instanceOf Connection) ports += '''
4    accepts input on <<connectionName.toFirstUpper>>
          with type <<dataReceived.type>>!'''
5
6    if(e instanceOf Action) transitions += '''
         passivate in s<<fromState>>!
7    when in s<<fromState>> and receive <<
         dataReceived>> go to s<<toState>>!'''
8
9  }else if((connection.type == OUTPUT) or (action.
       type == SEND)){
10   if(e instanceOf Connection) ports += '''
         generates output on <<this.connection.
         typeName>> with type <<dataSent.type>>!'''
11
12   if(e instanceOf Action) transitions += '''hold
         in s<<fromState>> for time 1!
13   after s<<fromState>> output <<this.connection.
         typeName>>!
14   from s<<fromState>> go to s<<toState>>!
15   '''
16   }
17 }
```

Listing 3: Excerpt of transformation code in Xtend

The names of connections in SoSADL are preserved by the transformation. The code in lines 4 and 10 in Listing 3 use the connection name to create namesake ports in DEVS, as shown in lines 24-27 of Listing 2. Besides that, the type of data received or sent is used for typifying data that are allowed to be transmitted in a DEVS port (lines 4 and 10 in Listing 3). Furthermore, `receive` instructions in SoSADL create receive transitions, such as lines 33, 35, and 37 in Listing 2. In turn, `passivate` are instructions used to make the sensor wait for a data that will be received, such as in Lines 32, 34, and 36 in Listing 2. Besides, `send` instructions in SoSADL generate three lines of code in DEVSNL: one that holds in a particular state for 1 second (we established this time as a default), one that produces the output of some data, and another that performs the state transition to the next state (lines 12-14 in Listing 3).

C. Generating Coupled Models

Listing 4 shows the SoSADL code that specifies a concrete software architecture for the FMSoS. In particular, the architectural configuration of this SoS comprises four sensors, one gateway, and four transmitters (lines 4-12). The `bindings` (lines 13-23) specify how connections between constituents and mediators are established through gates, which compose the interface of the software architecture of this SoS and define their dynamics for transmitting data among sensors towards a gateway station that is able to process them. A sensor collects the water level through actuators, encapsulating it with the specific location where it was obtained and its corresponding time stamp. After that, sensors can transmit observations to the closest transmitter[9], which forwards them to a nearby sensor or gateway. Following, additional details about the content of a binding block are presented.

```
1  sos FloodMonitoringSos is {
2    architecture FloodMonitoringSosArchitecture() is
         {
3      behavior coalition is compose {
4      sensor1 is Sensor
5      sensor2 is Sensor
6      sensor3 is Sensor
7      sensor4 is Sensor
8      gateway is Gateway
9      mediator1 is Mediator
10     mediator2 is Mediator
11     mediator3 is Mediator
12     mediator4 is Mediator
13     } binding {
14       relay gateway::notification::alert to
             warning::alert and relay gateway::
             request to request and
15       unify one { sensor1::measurement::measure }
             to one { mediator1::fromSensors } and
16       unify one { mediator1::transmit::
             towardsGateway } to one { sensor2::
             measurement::pass } and
17       unify one { sensor2::measurement::measure }
             to one { mediator2::fromSensors } and
18       unify one { mediator2::transmit::
             towardsGateway } to one { gateway::
             notification::measure } and
19       unify one { sensor3::measurement::measure }
             to one { mediator3::fromSensors} and
20       unify one { mediator3::transmit::
             towardsGateway } to one { sensor4::
             measurement::pass } and
21       unify one { sensor4::measurement::measure }
             to one { mediator4::fromSensors} and
22       unify one { mediator4::transmit::
             towardsGateway } to one { gateway::
             notification::measure }
23 } }
```

Listing 4: Description of an architecture of an FMSoS in SoSADL.

As shown in this listing, SoSADL specifies a connection in the form `system :: gate :: connection`. Indeed, the same gate can hold one or more connections. For each pair of sensors with a transmitter between them, an unification is established by an `unify` statement (lines 15-22). These statements specify that an output connection `measure` from the gate `measurement` of a particular sensor is linked to the input connection called `fromSensors` of the closest transmitter; and that a transmitter gathers

[9]This part of the code is specified in an atomic model which represents the operation of the constituent. These details will be presented in a forthcoming paper.

data from one sensor (lines 11-14) and forwards it to the next sensor. Transmitters have an output connection named `towardsGateway`, which are linked to sensors through an input connection called `pass` of gate `measurement`. This enables them to receive data for being transmitted and forward them to a gateway (lines 16, 18, 20, and 22). Lines 22 and 23 link the output connection of transmitters to a gateway connection, which is named `measure`. In this case, a transmitter mediates a sensor and a gateway. The `relay` statement (line 14) establishes the communication between the FMSoS and external systems, connecting the notification gate of a gateway to one external connection.

Listing 5 presents transformation rules that translate a software architecture expressed in SoSADL into a coupled model in DEVS. It depicts three transformation rules: one that receives a SoSADL type called `ArchitectureDecl` as input (lines 1-4), one that receives an `ArchBehaviorDecl` as input (lines 5-16), and one that compiles `Unify` statements (lines 17-40). The first transformation rule is responsible for producing the first line of the DEVS code which declares a Decomposition. This rule assumes the name of the architecture, and passes the remaining part to the next transformation rules[10]. In the second transformation rule, a list of the systems that compose the architecture is enumerated in the DEVS target model, hence completing the DEVS decomposition. Bindings are compiled in the next transformation rule that begins in line 17. The compilation of unifications, i.e., the specification of the data exchanged between systems and mediators in a coalition proceeds as follows. The sender and the receiver names are required for documenting the communication between systems in DEVS. Thus, the names are separated from data type in unifications, using the `::` as a marker to split the String (line 30). However, it is also necessary to know data types that are being transferred between two elements in the coalition for creating the simulation code. This data is not available in the architecture specification in SoSADL, but it is available in the specification of the constituents and mediators in SoSADL (hidden here due to lack of space).

```
1  def compile(ArchitectureDecl a) {
2    var String result  = '''From the top perspective
         , <<a.name.toFirstUpper>> is made of <<a.
         behavior.compile>>'''
3    result
4  }
5  def compile(ArchBehaviorDecl a){
6    var int size = a.constituents.size
7    var int cont = 0
8    var result  = ''''''
9    for (Constituent c: a.constituents){
10     cont++
11     result += '''<<IF (cont == size)>> and <<c.
           name.toFirstUpper>><<ELSE>><<c.name.
           toFirstUpper>>, <<ENDIF>>'''
12   }
```

[10]These transformation rules were structured as presented for separation of concerns, reuse, and modularization purposes

```
13   result +='''!'''
14   result += '''<<a.bindings.compile>>'''
15   result
16 }
17 override def compile(Unify u){
18   var String sender = u.connLeft.compile.toString
         ()
19   val String[] vector = sender.split('::')
20   var int firstIndex = sender.indexOf("::")
21   var String connectionSender = sender.substring(
         firstIndex+2,sender.length)
22   sender = vector.get(0)
23
24   var String receiver = u.connRight.compile.
         toString()
25   val String[] vector2 = receiver.split('::')
26
27   receiver = vector2.get(0)
28   var String data = "";
29   //Hidden code that reads connections data types
         from a file
30   val String[] vectorConnections = data.split("-")
31   for(String s: vectorConnections){
32     val String[] vectorAux = s.split(";")
33     var String connectionName = vectorAux.get(0).
           replace(" ","")                if(
           connectionSender.compareTo(connectionName)
           ==0){
34     data = vectorAux.get(1).toFirstUpper
35   } }
36   var String result = '''
37   From the top perspective,  <<sender.toFirstUpper
         >> sends <<data>> to <<receiver.toFirstUpper
         >>!
38   '''
39   result
40 }
```

Listing 5: Transformation rule from a SoSADL model to a DEVS model in Xtend.

Still in Listing 5, a portion of code that opens a file in which there is a specification of the connections and their respective data types was hidden for space reasons. The code in lines 31 to 35 compares the name of each pair gate-connection to the gates and connections specified in the coalition, inferring the type of data that they transmit. This data is assigned to the variable `data` when it is found (Line 34). Line 36 shows the format of the output String, with sender, receiver, and data, and Line 39 prints the result. The transformation rule for `Unify` is called once for each unification in the SoSADL specification. Each binding specified in SoSADL is mapped into one coupled model in DEVS.

Listing 6 shows the equivalent code derived for the architecture of the FMSoS. Line 1 shows that the `FloodMonitoringSoSArchitecture` is formed by the same systems specified in the SoSADL code. Lines 2 to 9 show data exchange between the systems and mediators derived from the SoSADL specification. Each line depicts a unification statement of the bindings block in the SoSADL model. Finally, the DEVS tool converts that code into an executable simulation model.

```
1  From the top perspective,
       FloodMonitoringSosArchitecture is made of
       Sensor1, Sensor2, Sensor3, Sensor4, Gateway,
       Mediator1, Mediator2, Mediator3,  and
       Mediator4!
2  From the top perspective, Sensor1 sends Measure to
        Mediator1!
3  From the top perspective, Mediator1 sends Measure
```

```
      to Sensor2!
4  From the top perspective, Sensor2 sends Measure to
       Mediator2!
5  From the top perspective, Mediator2 sends Measure
      to Gateway!
6  From the top perspective, Sensor3 sends Measure to
       Mediator3!
7  From the top perspective, Mediator3 sends Measure
      to Sensor4!
8  From the top perspective, Sensor4 sends Measure to
       Mediator4!
9  From the top perspective, Mediator4 sends Measure
      to Gateway!
```

Listing 6: Coupled Model for FMSoS generated in DEVS.

Moreover, Listing 6 shows that sensors transmit their data to the closest transmitter (Lines 2, 4, 6, and 8). Then, transmitters forward these data in Lines 3, 5, 7, and 9 to a nearby sensor or gateway. Since `Sensor2` and `Sensor4` send data to `Mediator2` and `Mediator4` respectively (lines 4 and 6), the gateway is already reached (lines 5 and 9). When these data arrive in the gateway, their values are tested against a predetermined depth threshold. If they are higher, the gateway behavior of sending a flood alert is triggered. Hence, this architectural configurations can successfully accomplish the SoS mission that is reliably producing flood alerts.

IV. Evaluation

ASAS was evaluated in a case study on a Flood Monitoring SoS (FMSoS), which is a SoS that is intended to be part of a smart city. FMSoS monitors rivers crossing urban areas, which pose great danger in rainy seasons, potentially damaging property, threatening lives, and spreading disease. FMSoS notifies possible emergency situations to residents, businesses owners, pedestrians, and drivers located near of the flooding area, and also to governmental entities and emergency systems. FMSoS is composed of five different types of constituents, as illustrated in Figure 1:

1) **smart sensors**, which are fixed embedded systems monitoring flood occurrences in urban areas, located on river edges;
2) **gateways**, which gather data from constituents and share them with other systems;
3) **crowd-sourcing systems**, which are mobile applications used by citizens for real-time communication of water level rising; danger level is a pre-defined value (between 1 and 6, 1 being no risk, and 6 being flood effectively occurring) that can be classified by the human user according to what he/she observes;
4) **drones**, which are UAVs also used to complement sensors observations by monitoring the river water level while they fly over it, sending pictures if some change in the water level occurs; and
5) **drone bases**, which are fixed basis from where drones departure, and to where they come back for battery recharging, and data transmission.

Moreover, it is intended the FMSoS be part of a larger SoS composed of Wireless River Sensors, Telecommunication Gateways, Unmanned Aerial Vehicles (UAVs), Vehicular Ad Hoc Networks (VANETs), Meteorological Centers, Fire and Rescue Services, Hospital Centers, Police Departments, Short Message Service Centers and Social Networks, as described in [15]. Such SoS involves the National Center for Natural Disaster Monitoring, which monitors 1000 cities, with 4700 sensors, including 300 hydrological sensors, and 4400 rain gauges.

To investigate the reliability of ASAS approach, we performed a case study, i.e., an exploratory type of empirical method for investigating a phenomena in its natural environment using data gathered from few entities (people, organizations, and sensors) [16]. It refers to a contemporary phenomenon observed in its real-world context. We evaluated our approach considering two cases, in the context of the FMSoS, that aims: (i) to investigate the accuracy of the transformation to create a valid simulation model that represents functional capabilities of the FMSoS (i.e., flood monitoring), and (ii) to investigate if the generated simulation model supports modifications in the FMSoS software architecture at runtime, and if with such model it is possible to evaluate different configurations for the FMSoS architecture. Each case is explained in the remaining of this section detailing the research questions that each case must resolve, the rationale for each question, the metrics used to answer such questions, collected data and procedures for data analysis and a discussion of results for each case.

A. Case 1: Accuracy of Simulation Model of the FMSoS Architecture

We specified one FMSoS architecture with 42 sensors, 9 crowd-sourcing systems, and 18 drones, following the model shown in Figure 1. Each drone has its own base (18 drone bases), and transmits the information collected through a gateway that will be in the vicinity. 18 gateways are spread along the river boards. Mediators were produced as much as necessary to mediate these constituents. FMSoS is concerned with a single behavior: *flood alert*.

RQ1. Was the transformation successful?

Rationale. Since the simulation model is automatically generated, it is important to check the validity of the produced model. A transformation can be considered successful if the simulation runs without errors.

M1. Simulation failures: given by the quantity of detected failures during model simulation, such as simulation crashing or stopping.

RQ2. How accurate was the simulation in supporting the monitoring of an SoS operation?

Rationale. This question addresses whether or not the simulation is capable of reproducing SoS behavior, monitoring it and guaranteeing that it can be considered trustworthy.

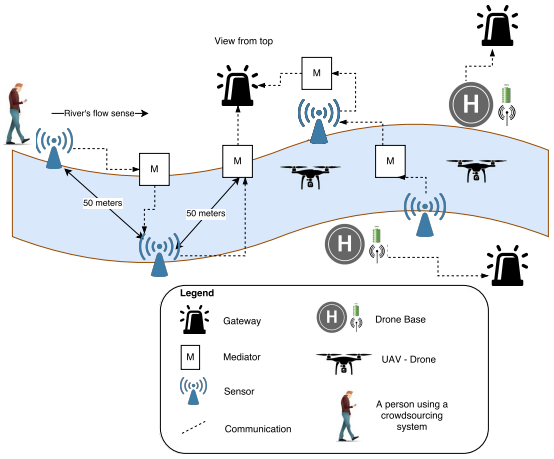

Figure 1: A Flood Monitoring System-of-System (FMSoS) as part of a Smart City System.

M2. Accuracy: measured by the proportion of alerts correctly triggered by the FMSoS.

Procedures for the analysis of collected data.

To analyze our data, we chose the following indices [12]:
True Positive (TP) (equivalent to the hit): when a flood alert in the simulation represents an actual real flood;
True Negative (TN) (equivalent with correct rejection): when the alarm was not triggered in the simulation, as it was not triggered in reality, since there was not a flood;
False Positive (FP) (false alarm, known as Type I error): when it was triggered, but there was not a flood. This can happen, for instance, due to accumulation of sediment within the sensitivity radius of the actuator;
False Negative (FN) (miss, known as Type II error): when there was a flood, and the alarm was not triggered.

We also defined supplementary metrics related to RQ2, which are [12]:
Sensitivity (S): proportion of positives that are correctly identified as such;
Specificity (Sp): proportion of negatives that are correctly identified as such;
Accuracy (A): relation between the correct diagnostics (TP + TN) and the total amount;
Precision (P): relation between the true positives and all the positives (true and false).

These supplementary metrics are computed as:

$$S = \frac{TP}{TP+FN} \quad Sp = \frac{TN}{FP+TN}$$
$$P = \frac{TP}{TP+FP} \quad A = \frac{TP+TN}{TP+TN+FP+FN}$$

Data used. We chose a dataset collected by the actual FMSoS over four days, from November 23th 2015 to November 27th 2015. This interval was important because during these months a number of floods occurred. This enabled us to establish whether or not our simulation results in a diversity of situations. We established *a 4-window strategy* implemented at the gateways that receive data from constituents to confirm floods. For each set of four data that subsequently arrives, the gateway checks them. For the period studied, the river had an average rise from 35 to 50 cm, depending on the location. Thus, in this context, the threshold of a flood is defined as a rise of 100 cm or more. If at least one pair of data that arrived have both their depth levels at least 100 cm (the threshold established for that city), a flood alarm is triggered. Table III illustrates a numerical instance. It corresponds to real data that arrived sequentially at the gateway. Data that arrive are chronologically ordered, and pairs of data are analyzed. If at least one pair has two measures equal or greater than 100 cm, a flood is confirmed. Subsequent measures will confirm if it is an actual flood or not.

Table III: A sample of data collected by a sensor and sent to a gateway.

sample id	sensor	timestamp	depth (cm)
#1	S2	2015-11-23 01:58	58

Data were stored in text files and delivered by the stimuli generators along the FMSoS, feeding the simulation. These stimuli generators delivered 1,000 samples for each sensor. Timestamps represented that each data sample was sent every five minutes for each sensor (i.e., 12 samples by hour, 288 per day, totalizing 3,47 days of data simulated). We also considered an amount of 1,000 samples for each crowd-sourcing system that is part of the SoS. We adapted our dataset so to have similar data for stimuli generators for crowd-sourcing systems and drones. For crowd-sourcing systems, the aforementioned scale was used to classify risk between 0 and 6. Zero means that crowd-sourcing systems is not contributing to flood diagnosis. We mapped the height of water in original data to a scale of risk between 1 and 6, 1 being no risk, and 6 being flood effectively occurring. Human users can classify a risk between these values according to what he/she sees. So we could imitate how people would react and behave according to the changes in water level registered before by sensors. Then, we created a dataset corresponding to the data used to feed sensors. For drones, we used 5,000 data per drone. Each drone has an autonomy to fly 2,500 meters and return to the basis. This flight usually lasts five minutes, monitoring the same portion of the river, and returning to recharge and transmit data. We set up 18 drones to monitor the entire extension of the river (45 kilometers). The drone flies 2500m in five minutes, that is, 500 meters per minute. This makes one measurement per minute. As this is done 1,000 times, it

corresponds to 5,000 samples by drone. So drones have five times more data than sensors (90,000 samples). Each drone data sample was delivered every five minutes, totalizing this amount for the entire days that we consider in our sample. Then, we worked on a set of 141,000 data samples (42,000 for sensors, 9,000 for crowd-sourcing system, and 90,000 for drones). As the SoS has 69 constituents and 18 gateways, an average of almost four constituents are connected to each gateway. Any gateway can deliver a flood alert individually, whilst the existence of a flood will be confirmed by the subsequent flood alert given by other gateways due to the increase in the water level.

Reporting on Case 1. Day November 23th was the most relevant day. Other days exhibited levels higher than 100 cm, but only as a momentary occurrence. Stimuli generators were capable of delivering the data throughout simulation execution. As data were distributed equally between each gateway and its nearest neighbors, around 7,833 samples (141,000 by 18 gateways) were transmitted by each gateway during the entire simulation. As each gateway data group used the four-window strategy, around 1,958 groups of data were analyzed by each gateway in order to ascertain the existence of true or false positives or negatives. Next we discuss the answers to the research questions.

RQ1. Was the transformation successful?
The simulation run accordingly with no failures. Thus, we can consider that the transformation was feasible and well-succeeded for this particular context. Further applications should be tested. For now, M1 is equals to 0.00%.

RQ2. How accurate was the simulation in supporting the monitoring of an SoS operation?
The simulation took six hours and 20 minutes to run on an Intel core i5-3230M 2.60GHz (x64) processor, with 4 GB of RAM Memory, 1TB of HD, and running Ubuntu 16.04 with 64 bits. The data corresponds to four days of monitoring data from a specific real river. This step was evaluated according to the metrics established in RQ2. 16 true positives occurred, since during the considered period, besides one effective flood (November 23th), in which the level of water reached almost 7 meters. A total of 1,942 true negatives correctly arise. Nor false positives nor false negatives occurred. The river that crosses the city has an average level of 35 to 50 centimeters (cm), depending on the location. For this context, the threshold of a flood is considered about 100 cm. For all situations in which a real condition of threat of flood occurred, the flood alert emerged as a result of the constituents interoperability. Then, at least in this case study, the Sensitivity (S), Specificity (Sp), Accuracy (A), and Precision (P) are all equal to 100%. The flood alert was triggered with high accuracy, conforming to the pre-established purposes of the investigation. Therefore, our results imply a high level of confidence and feasibility.

B. Case 2: Analysis of FMSoS Dynamic Architecture

Along the SoS runtime, constituents can join or leave the SoS, raising different architectural configurations (coalitions). The purpose of Case 2 is twofold: (i) to assess whether our approach enables to transform, simulate, and evaluate a SoS software architecture specification by comparing different architectural configurations, and (ii) to evaluate if our approach supports simulation of SoS operation, considering its dynamic architecture, showing at runtime how the SoS software architecture evolves by the addition or removal of constituents. Based on this goal, the research question for this case is:

RQ3. Does ASAS support the simulation and evaluation of different architecture configurations for SoS software architecture? For this case, drones and drone bases were not considered. We defined the following variables to attempt to address RQ3: *scale* (S), i.e., number and diversity of constituents, *architectural configurations* (AC), to analyze which one offers best results, and the *number of external interfaces* (NEI), i.e., how many gateways interface with external systems.

Rationale. RQ3 is posed to investigate whether our approach supports the comparison of different architectural configurations leveraging SoS operation. It is hoped that this would enable one to select the configuration that best suits the needs of the SoS as well as defining custom strategies for maintaininging SoS quality, in spite of a changing operational environment.

Metrics: (i) the percentage of data samples that were correctly transmitted along each architectural configuration, and (ii) the percentage of flood alerts that were triggered (accuracy).

Data used. Data were provided by a real project that combine crowd-sourcing and sensor data for detecting floods [9]. We chose a comprehensive data sample collected from November 23th 2015 to November 27th 2015, a period in which there were intense rains and floods.

Simulation. We were able to analyze whether our simulation results conform to the expected FMSoS behavior, i.e., sending a flood alert whenever collected data exceeded a predetermined threshold. We analyzed 50 different architectural configurations, varying the number of sensors, crowd-sourcing systems, and gateways (as well as mediators dynamically appearing between constituents). All 50 architectural configurations were evaluated, starting with a configuration with four sensors, one gateway, and zero crowd-sourcing systems (besides the necessary mediators). Progressively, the number of sensors was increased, followed by the number of gateways and crowd-sourcing systems.

Analysis of collected data. Since results of this simulation are stored in log files, it was possible to calculate the corresponding value of the aforementioned metrics. After that, we compared these values to select configurations with

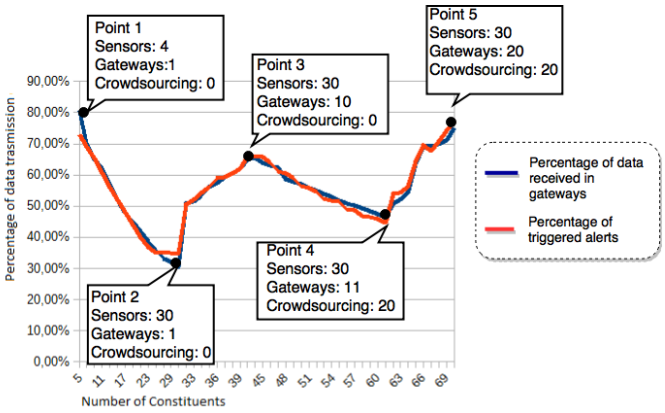

Figure 2: Relation between percentage of data received in gateways and alerts triggered.

the best results.

Reporting. Figure 2 summarizes the simulation outcome, taking into account (i) the percentage of the 1000 data samples fed to each of the sensors that were correctly transmitted along the SoS architecture until the gateways considering the variation in the number of constituents, and (ii) the percentage of flood alerts that were triggered. It was observed that data loss increased with the number of sensors, reducing both the reliability of data transmission and triggered alerts (Point 2). This loss was alleviated by increasing the number of gateways, which increased the numbers of transmission rate and triggered alerts. When the architecture configuration had 40 constituents (Point 3), i.e., 30 sensors and 10 gateways (without considering mediators), the number of crowd-sourcing systems was increased as well. However, despite the expectation, increasing the number of crowd-sourcing systems did not increase the transmission rate, nor the number of alerts triggered because of the bottleneck of the gateways. Results improved again when the number of crowd-sourcing systems was fixed at 20 (Point 4), and the number of gateways was increased to 20 (Point 5), with 30 sensors, 20 gateways, and 20 crowd-sourcing systems (70 constituents, without considering mediators). It was also possible to observe that the rate of alerts correctly triggered was close to the rate of data effectively transmitted.

Good results occurred when FMSoS has many constituents, but these results are not better than when FMSoS has only five constituents. Hence, unless there is a situation in which a geographic area to be covered is too large, using a small number of constituents can achieve the same results than using a large number, at least for this domain, these configurations defined, and these types of constituents. Finally, we conclude that our approach successfully supported the automatic generation of simulation models for FMSoS specifications, and facilitated the evaluation of different architectural configurations.

Discussion. We concluded that for the improvement of FMSoS operation, gateways were the critical element to improve its operation. Moreover, the use of four sensors is as good as 70 constituents. This result is also important because it enables architects to spend significantly less effort and money by using only a few sensors to develop software and devices to design such an SoS. It would not have been possible to draw this conclusion at the design stage without anticipating the benefits that ASAS provides. Moreover, data loss happens in simulation because network services that guarantee data delivery (such as TCP) are not natively implemented in simulation formalisms. As SoS dynamic architectures enable changing and to diversifying the number of constituents, the solution generated by our proposed approach can be easily scaled, as shown in our case study. DEVS does not hold an specific data type for the environment modeling in its syntax. Moreover, all constituents are modeled as a same type of model (atomic model) in DEVS, reducing precision in architectural description. SoSADL, in turn, overcomes all the aforementioned DEVS drawbacks, except for simulation [14]. We also claim that we contribute to productivity and reuse in SoS engineering. Programming the model transformation to automatically produce simulation models by one specialist with integral dedication took around four months of work. Despite the learning curve associated with DEVS modeling, Xtend programming, and domain-specific knowledge to adapt model transformations, the model transformation can be reused in a myriad of other domains. Otherwise, producing simulation models for each type of constituent of an SoS and for the SoS architecture itself can take a similar amount of time, besides the time to model the SoS itself. Hence, this effort can be reduced to the specification of SoS models by using our approach.

C. Threats to Validity

We can mention some threats to the validity for Case 1: the scale of our evaluation, the verification of correctness of the transformation rules, and the topology of the SoS. Our solution can scale, as scaling SoS consists of the specification of further bindings in the coalitions in SoSADL and replications of atomic models in DEVS. Regarding transformation correctness, we established correspondences between entities in both models and the resulting simulation model relieves the threat, showing a solution. Further investigations of topologies and different number of constituents will be carried out. Due to the limited time window and the suitability of the period for our purposes, the selection of the data might present a possible bias. However, the limited period has a plurality of inputs.

For Case 2, we identified the following threats to our study: (i) data selected to feed the simulation: Data were selected from a real case conducted in a smart sensors network and a crowd-sourcing system deployed in an actual city. We merged these data during SoS operation by feeding the respective constituents to the stimuli generators. The 1000

observations were collected during four days encompassing flooding occurrences. Hence, this threat was mitigated; (ii) scaling: Scale is not a problem since our simulation can handle a larger number of constituents. However, this hampers data visualization and processing, as a large number of constituents is difficult to visualize in a simulation and a more powerful processor would be required to execute simulation of a larger SoS. However, these problems are reduced as the resulting data are saved in spreadsheets to be properly analyzed; (iii) the order in which we performed changes in the architecture and the set of changes that we proposed to perform on the architecture: Different changes could be performed, substituting or eliminating constituents, or even reorganizing SoS configuration. However, the purpose of this study was to check if our approach could successfully support the automatic generation of functional simulation models and the analysis of different architectural configurations according to predetermined metrics. Then, this threat did not influence our results. Further investigation about other dynamic architecture operators must be conducted, in particular, related to substitution and deletion of constituents and architecture reorganization as well.

V. Final Remarks

This paper presented ASAS, an approach that automatically generates simulation models by means of a model transformation from SoS architecture descriptions expressed in SoSADL to DEVS. Our approach supports the analysis of alternative architectural configurations at runtime, thus enabling to leverage SoS operation by identifying best architectural configurations. Simulation models can certainly vary depending on which systems constitutes the SoS. Indeed, our approach is domain-independent. Hence, we claim that ASAS can be adopted to transform static models represented in SoSADL to dynamic simulation models of SoS.

Future work might include (i) reproducing the case study in other scenarios, (ii) developing a deeper understanding of the results achieved, especially with different types of constituents; (iii) investigating if the ratio between the number of gateways and the number of constituents influences the data transmission ratio; and (iv) developing artificial intelligence and inference mechanisms (such as neural networks) to trainee the SoS and teach it how to maintain beneficial architectural configurations identified with the ASAS approach. We claim that the ASAS approach can be adopted as a platform to represent different facets of a smart city and other SoS applications, such as coupling distinct models (such as a transportation model and a building energy demand model), establishing new interoperability links to achieve and offer more complex solutions. Other domains can benefit of our approach, as SoS can be modeled in SosADL and automatically transformed to simulation models. Potential domains include health care smart systems, smart buildings, smart farms, aerospace exploration SoS, and autonomic cars.

Acknowledgments

This work is supported by São Paulo Research Foundation (FAPESP), grants 2012/24290-5, 2013/20317-9, 2014/02244-7, and 2017/06195-9. We also thank professor Les Foulds (INF-UFG) by the precious advises and language review.

References

[1] B. Boehm. “A View of 20th and 21st Century Software Engineering”. In: ICSE ’06. Shanghai, China: ACM, 2006, pp. 12–29.

[2] E. Cavalcante, F. Oquendo, and T. V. Batista. “Architecture-based code generation: from π-ADL architecture descriptions to implementations in the Go language”. In: *ECSA*. Vienna, Austria, 2014, pp. 130–145.

[3] E. Cavalcante et al. “Statistical Model Checking of Dynamic Software Architectures”. In: *ECSA*. Copenhagen, Denmark, 2016, pp. 185–200.

[4] K. Falkner et al. “Model-driven performance prediction of systems of systems”. In: *Software & Systems Modeling* (2016), pp. 1–27.

[5] B. B. N. de França and G. H. Travassos. “Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines”. In: *Empirical Software Engineering* 21.3 (2016), pp. 1302–1345.

[6] V. V. Graciano Neto. “Validating Emergent Behaviors in Systems-of-Systems through Model Transformations”. In: *ACM Student Research Competition at MODELS*. Saint Malo, France, 2016, pp. 1–6.

[7] V. V. Graciano Neto. “A Model-Based Approach Towards the Building of Trustworthy Software-Intensive Systems-of-Systems”. In: *ICSE Companion 2017*. Buenos Aires, Argentina: IEEE, 2017, pp. 425–428. DOI: 10.1109/ICSE-C.2017.28.

[8] M. Guessi et al. “A systematic literature review on the description of software architectures for systems of systems”. In: *SAC*. Salamanca, Spain, 2015, pp. 1433–1440.

[9] F. E. Horita et al. “Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks”. In: *Computers & Geosciences* 80 (2015), pp. 84 –94.

[10] M. Jamshidi. “System of Systems - Innovations for 21st Century”. In: *ICIIS*. 2008, pp. 6–7.

[11] M. W. Maier. “Architecting principles for systems-of-systems”. In: *Systems Engineering* 1.4 (1998), pp. 267–284.

[12] R. Malhotra. *Empirical research in software engineering: concepts, analysis, and applications*. CRC Press, 2016.

[13] C. B. Nielsen et al. “Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions”. In: *ACM Comput. Surv.* 48.2 (Sept. 2015), 18:1–18:41.

[14] F. Oquendo. “Formally Describing the Software Architecture of Systems-of-Systems with SosADL”. In: *SOSE*. Kongsberg, Norway, 2016, pp. 1–6.

[15] F. Oquendo. “Software Architecture Challenges and Emerging Research in Software-Intensive Systems-of-Systems”. In: *ECSA*. Copenhagen, Denmark, 2016, pp. 3–21.

[16] P. Runeson and M. Höst. “Guidelines for Conducting and Reporting Case Study Research in Software Engineering”. In: *Empirical Softw. Engg.* 14.2 (Apr. 2009), pp. 131–164. ISSN: 1382-3256.

[17] D. Wachholder and C. Stary. “Enabling emergent behavior in systems-of-systems through bigraph-based modeling”. In: *SOSE*. San Antonio, USA, 2015, pp. 334–339.

[18] G. Wiederhold. “Mediators in the architecture of future information systems”. In: *Computer* 25.3 (1992), pp. 38–49.

[19] X. Xia et al. “A Model-Driven Approach for Evaluating System of Systems”. In: *ICECCS*. Singapore, 2013, pp. 56–64.

[20] B. P. Zeigler et al. *Guide to Modeling and Simulation of Systems of Systems*. Springer, 2012.